```
/*
        Fog  Filter
        Copyright July 2003 4G Color
        All rights reserved
        GD 7.18.03


        First approach is just to convert the PA to the two plists...
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define FORM  "4gFilterForm"
#define PARAM "4gParamArray"
#define FILENAME "4gParamArrayTest"

typedef unsigned char U8 ;
typedef short   UVAL;       // uval is typically (-100,100)

enum colorDef { red, yellow, green, cyan, blue, magenta, neutral};
const int nColors = 7;
const int versionCode = 3314;

// filter structure, size is 2*((9*6)+4) = 116 bytes
typedef struct
{
    // neutral axis (NP) color correction
    UVAL    whiteNP         [nColors];  // (+,-)
    UVAL    lightGrayNP     [nColors];  // (+,-)
    UVAL    darkGrayNP      [nColors];  // (+,-)
    UVAL    blackNP         [nColors];  // (+,-)
    // neutral brightness correction
    UVAL    lightGray;                  // (whiter,grayer)
    UVAL    darkGray;                   // (blacker,grayer)
    // saturated color correction
    UVAL    colorShift      [nColors];  // (r>y>g>c>b>m>r,m<r<y<g<c<b<m)
    // tone correction
    UVAL    brightColors    [nColors];  // (moreVivid,whiter)
    UVAL    deepColors      [nColors];  // (moreVivid,blacker)
    UVAL    midToneColor    [nColors];  // (moreVivid,grayer)
    UVAL    midToneLightness[nColors];  // (lighter,darker)
    // acuity
    UVAL    texture;                    // (sharper,smoother)
    UVAL    edges;                      // (sharper,smoother)
} filterDEF;

void writeFilterForm(FILE *pForm, filterDEF fx);
void scanFilterForm(FILE *pForm, filterDEF *fx);
void initFilter(filterDEF *fx);
void printFilter(filterDEF fp);

typedef enum   {inscribe, superscribe, anamorphic}                  cropDEF ;
typedef enum   {fA, fB, AplusB, ABblend}                            compositionDEF;
typedef enum   {autoExposureOff, autoExposure100, autoExposure80}   autoExposureDEF;
typedef enum   {autoColorBalanceOff, autoColorBalanceOn}            autoColorBalanceDEF;
typedef enum   {jpegOff, jpegOn}                                    jpegFilterDEF;

// process structure
typedef struct
{
    // resize, reshape, and crop
    long    inPixels;
```

ATTACHMENT 9

```
    long    inLines;
    long    outPixels;
    long    outLines;
    cropDEF cropSelect; // (inscribe, superscribe, anamorphic)
    // compensation
    autoExposureDEF     autoExposureSelect;      // {autoExposureOff, autoExposure100, autoExp
    autoColorBalanceDEF autoColorBalanceSelect;  // {autoColorBalanceOff, autoColorBalanceOn}
    jpegFilterDEF       jpegFilterSelect;        // {jpegOff, jpegOn}
    // filter composition and definition
    compositionDEF  filterSelect;        // {fA, fB, AplusB, ABblend}
    short           filterAgain;         // (+,-)
    short           filterBgain;         // (+,-)
    filterDEF       filterA;
    filterDEF       filterB;
} fogParamArrayDEF;


#if 0
enum toneDef
{                       //  allowed changes                 variable name
    brightColors,       //  (moreVivid, whiter)         //  10  uVal[7][0]
    deepColors,         //  (moreVivid, blacker)        //  11  uVal[7][1]
    midToneColor,       //  (moreVivid, grayer)         //  12  uVal[7][2]
    midToneLightness,   //  (lighter,   darker)         //  13  uVal[7][3]
    colorShift,         //  (r>y>g>c>b>m,  r<y<g<c<b<m) //  14  uVal[7][4]
    whitePoint          //                              //  15  uVal[7][5]
};

enum neutralDef
{
    lightGrayNeutrals,  //  (whiter,    grayer)         //  16  lGray
    darkGrayNeutrals,   //  (blacker,   grayer)         //  17  dGray
    spatialAcuity,      //  (sharper,   smoother)       //  18  acuity
    filterGain          //  (positive,  negative)       //  19  fgain
};

#endif
class image
{
    private:    // image access
                U8      *pSource, *pTarget;
                int     width, height, offset, plates ;
                // master transform generation
                long    ffTable[256], iiTable[256], rrTable[256];
                long    *v0, *v1, *v2, *v3, *v4, *v5, *v6, *v7, *v8;
                long    *b0, *b1, *b2, *b3, *b4, *b5, *b6, *b7, *b8;
                // transform parameters
                fogParamArrayDEF pp;
                //int   autoExposure, foreground, compose, reSize[4], format, pack, diags, vC
                //int       uVal[7][10], lGray, dGray, acuity, fGain, plist[7][10];
                //int       uVal1[7][10], lGray1, dGray1, acuity1, fGain1, plist1[7][10];
                int     ppf[9], ppl[7][9];
                int     rwp, gwp, bwp;
                int     rwpA, gwpA, bwpA;
                int     rwpB, gwpB, bwpB;
                // selection parameters
                long    shsm, fore;
                // image statistics
                long    dist[256], qist[256], hist[256], nHist;
    public:             image( U8 *pS, U8 *pT, long w, long h, long o, long p );
                void    transform();
                void    getStats();
                void    showTable();
                void    initTables();
```

```
                void    permute(long *p, long *x, long *y, long *z, long r, long g, long b);
                void    unPermute(long *r, long *g, long *b, long p, long x, long y, long z);
                long    zcalc(long xx, long yy, long x, long y, long z, long U, long *p);
                void    neighborhood(long *R, long *G, long *B, long pixels, long lines);
                void    pixelTransform(long *xx, long *yy, long *zz,long *p, long x, long y, l
                void    setParameters( long x, long y, long z, long p );
                void    setRerangeTable();
                long    cubicTransform(long a, long b, long c, long d, long m0, long m1, long
                long    cubic(long m0, long m1, long x);
                void    transformConversion( long u, long *inputTable, long *ouputTable );
                void    rerange(long *x, long *y, long *z);
                long    tt( long uk, long uw, long x );
                long    sel( long k, long w, long x);
                long    ttt( long u, long x );
                void    setFore(int pixels, int lines);
                void    whitePointShift();
                void    initUserVals();
                void    readProcess();
                void    translate();
                void    writeForm(FILE *pForm);
                void    scanForm(FILE *pForm);
};

// Image Constructor
image::image( U8 *pS, U8 *pT, long wi, long hi, long of, long pl )
{
    // initialize parameters, essentially the api interface
    pSource = pS; pTarget = pT; width = wi; height = hi; offset = of; plates = pl;

    // allocate memory for 9 tables, and assign pointers
    b0 = (long *)malloc(9*256*sizeof(long));
    b1 = b0 + 256;
    b2 = b1 + 256;
    b3 = b2 + 256;
    b4 = b3 + 256;
    b5 = b4 + 256;
    b6 = b5 + 256;
    b7 = b6 + 256;
    b8 = b7 + 256;

    v0 = (long *)malloc(9*256*sizeof(long));
    v1 = v0 + 256;
    v2 = v1 + 256;
    v3 = v2 + 256;
    v4 = v3 + 256;
    v5 = v4 + 256;
    v6 = v5 + 256;
    v7 = v6 + 256;
    v8 = v7 + 256;

    // fixed tables are used for several basic functions
    initTables();

}


// permutations for primary and secondary colors...
// color        primary (pf)    secondary   secondary function (sf)
//  0 red       0 red           2 green     1 yellow
//  1 yellow    2 green         0 red       1 yellow
//  2 green     2 green         4 blue      3 cyan
//  3 cyan      4 blue          2 green     3 cyan
//  4 blue      4 blue          0 red       5 magenta
//  5 magenta   0 red           4 blue      5 magenta
```

```
const long pf[]     = {0,2,2,4,4,0};
const long sf[]     = {1,1,3,3,5,5};
const long pOrder[] = {1,2,3,4,5,0};
const long nOrder[] = {5,0,1,2,3,4};
const long sign[]   = {1,-1,1,-1,1,-1};
const long next1[]  = {1,0,3,2,5,4};
const long next0[]  = {5,2,1,4,3,0};


// unit interval cubic curve generator
// y = m0x + (3-2m0-m1)xx + (m0+m1-2)xxx
long image::cubic(long m0, long m1, long x)
{
    return (x*((255*255*m0 + (3*255-2*m0-m1)*255*x + (m0+m1-2*255)*x*x)/(255*255)) + 254)/255;
}

// cubic curve over intervals x=(a,b), y =(c,d)
long image::cubicTransform(long a, long b, long c, long d, long m0, long m1, long x)
{
    x = (255*(x-a))/(b-a);
    m0 = (m0*(b-a))/(d-c);
    m1 = (m1*(b-a))/(d-c);
    return (255*c + cubic(m0,m1,x)*(d-c))/255;
}

// selection correction approximates the yx to vx error
long image::sel( long k, long w, long x)
{
    if( w>k)    return (k*255 + (w-k)*( (x*x*(3*255 - 2*x))/(255*255) ))/255;
    else        return (k*255 - (w-k)*( (x*(6*x*255 - 4*x*x -3*255*255))/(255*255) ))/255;
}

long image::ttt( long u, long x )
{
    if( u>0 )   return (x*255 + u*(b2[x]-x))/255;
    else        return (x*255 - u*(b6[x]-x))/255;

}

void image::whitePointShift()
{
    long R = 255+rwp;
    long G = 255+gwp;
    long B = 255+bwp;

    long max = R>G ? R:G;
         max = B>max ? B:max;

    if (max <= 0 ) return;

    R = (255*R)/max;
    G = (255*G)/max;
    B = (255*B)/max;

    if ( R==0 || G==0 || B==0 ) return;

    for( int lines=0; lines<height; lines++ )
    for( int pixels=0; pixels<width; pixels++ )
    {
        // get the current pixel values
        long addr = pixels*plates + lines*offset;
        long r = pSource[addr+0];
        long g = pSource[addr+1];
        long b = pSource[addr+2];
```

```
        r = (R*r)/255;
        g = (G*g)/255;
        b = (B*b)/255;

        // now lets fix the brightness
        if( r==0 || g==0 || b==0 ) continue;
        long kr, kg, kb;

        // need to handle equalities as special cases!!!!
        if( r>=g && r>=b )
        {
            kr = R;
            kg = (r*G)/g;
            kb = (r*B)/b;
            if( kr<=kg && kr<=kb )
            {
                r = (r*255)/kr;
                g = (g*255)/kr;
                b = (b*255)/kr;
            }
            else if( kg<=kr && kg<=kb )
            {
                r = (r*255)/kg;
                g = (g*255)/kg;
                b = (b*255)/kg;
            }
            else
            {
                r = (r*255)/kb;
                g = (g*255)/kb;
                b = (b*255)/kb;
            }
        }
        else if( g>=r && g>=b )
        {
            kr = (g*R)/r;
            kg = G;
            kb = (g*B)/b;
            if( kr<=kg && kr<=kb )
            {
                r = (r*255)/kr;
                g = (g*255)/kr;
                b = (b*255)/kr;
            }
            else if( kg<=kr && kg<=kb )
            {
                r = (r*255)/kg;
                g = (g*255)/kg;
                b = (b*255)/kg;
            }
            else
            {
                r = (r*255)/kb;
                g = (g*255)/kb;
                b = (b*255)/kb;
            }
        }
        else
        {
            kr = (b*R)/r;
            kg = (b*G)/g;
            kb = B;
            if( kr<=kg && kr<=kb )
```

```
            {
                r = (r*255)/kr;
                g = (g*255)/kr;
                b = (b*255)/kr;
            }
            else if( kg<=kr && kg<=kb )
            {
                r = (r*255)/kg;
                g = (g*255)/kg;
                b = (b*255)/kg;
            }
            else
            {
                r = (r*255)/kb;
                g = (g*255)/kb;
                b = (b*255)/kb;
            }
        }
        // output RGB
        pSource[addr+0]   = (U8)r;
        pSource[addr+1]   = (U8)g;
        pSource[addr+2]   = (U8)b;
    }
}

void image::setFore(int pixels, int lines)
{
    if      ( foreground == 1 ) fore =  ((510*lines)/height) - 255;     // bottom
    else if ( foreground == 2 ) fore =  255 - ((510*pixels)/width);     // left
    else if ( foreground == 3 ) fore =  ((510*pixels)/width) - 255;     // right
    else if ( foreground == 4 ) fore =  ((510*lines)/height) - 255;     // center
    else if ( foreground == 5 ) fore =  ((510*lines)/height) - 255;     // center+bottom
    else                        fore =  255;        // do nothing

    // apply a S curve to fore here
}

// start ********* fundamental theorem for inverse parametric functions **************
// unfortunately, this takes 4KB of memory!
// derive the 9 basic transforms
void image::initTables()
{
    // spatial arrangement of indices
    // 0 1 2    -+  0+ ++
    // 3 4 5    -0  00 +0
    // 6 7 8    --  0- +-

    // first create the base function's in vu coordinates
    for( int i=0;i<256; i++)
    {
        long v = (255*i - i*i)/255; // gain is 255/270

        v0[i] = -(v*(255-2*i))/255;
        v1[i] =  (v*(255-i))/255;
        v2[i] =  v;
        v3[i] = -(v*i)/255;
        v4[i] =  0;
        v5[i] =  (v*i)/255;
        v6[i] = -v;
        v7[i] = -(v*(255-i))/255;
        v8[i] =  (v*(255-2*i))/255;

    }
```

```
    // convert to xx' coordinates
    transformConversion(255,  v0,  b0);
    transformConversion(255,  v1,  b1);
    transformConversion(255,  v2,  b2);
    transformConversion(255,  v3,  b3);
    transformConversion(255,  v4,  b4);
    transformConversion(255,  v5,  b5);
    transformConversion(255,  v6,  b6);
    transformConversion(255,  v7,  b7);
    transformConversion(255,  v8,  b8);

}

// note that the 45 degree translation causes jagged edges
// apply a post transform monotonic filter
void image::transformConversion( long u, long *inputTable, long *outputTable )
{
    for( int i=0, j=0, v;i<256; i++)
    {
        j=0;
        if( u*inputTable[i] >= 0 )
        {
            while ( ((u*inputTable[i+j])/255) > j && (i+j)<256) j++;
            v = i+(3*j)/2;
            v = v>255 ? 255:v;
        }
        else
        {
            while ( ((u*inputTable[i-j])/255) < -j && (i-j)>=0) j++;
            v = i-(3*j)/2;
            v = v<0 ? 0:v;
        }
        outputTable[i]=v;
    }
}

// interpolate the 9 basic transforms, piecewise bilinear interpolation
    // spatial arrangement of indices of the b-tables
    // 0 1 2    -+  0+ ++
    // 3 4 5    -0  00 +0
    // 6 7 8    --  0- +-
long image::tt( long k, long w, long x )
{
    long z0, zu, zv, zuv, u, v;
    z0 =  b4[x];
    if( k>0 )
    {
        zu =  b5[x];    u = k;
        if( w>0 )   {   zv =  b1[x];    zuv = b2[x];    v = w;  }   // upper right quad
        else        {   zv =  b7[x];    zuv = b8[x];    v = -w; }   //  lower right quad
    }
    else
    {
        zu =  b3[x]; u = -k;
        if( w>0 )   {   zv =  b1[x];    zuv = b0[x];    v = w;  }   //  upper left quad
        else        {   zv =  b7[x];    zuv = b6[x];    v = -w; }   //  lower left quad
    }
    return (z0*(255-u)*(255-v) + zu*u*(255-v) + zv*(255-u)*v + zuv*u*v)/(255*255);
}
// end ********* fundamental theorem for inverse parametric functions **************


// transform diagnostic graphs
void image::showTable()
```

```
{
    if( diags != 2 ) return;
    if( width<256 || height<256 ) return;
    // initialize with a generic linear transform
    long lin[256];
    for( int i=0;i<256; i++)    lin[i] = i;
    long  *zR=lin,*zR2=lin,*zG=lin,*zB=lin,*zC=lin,*zM=lin,*zY=lin,*zW=lin,*zW2=lin,*zW3=lin; /

    // create some test patterns
    long  aTab[256],bTab[256],cTab[256],dTab[256],eTab[256],fTab[256],gTab[256],hTab[256],iTab[

    long a = 255;
    for( int i=0; i<256; i++ )
    {
        aTab[i] = tt( a, a, i);
        bTab[i] = tt( a, 0, i);
        cTab[i] = tt( a,-a, i);

        dTab[i] = tt( 0, a, i);
        eTab[i] = tt( 0, 0, i);
        fTab[i] = tt( 0,-a, i);

        gTab[i] = tt(-a, a, i);
        hTab[i] = tt(-a, 0, i);
        iTab[i] = tt(-a,-a, i);
    }


    // the real assignments
    zR =  aTab;
    zG =  bTab;
    zB =  cTab;

    zC =  dTab;
    zM =  eTab;
    zY =  fTab;

    zW2 =  gTab;
    zW3 =  hTab;
    zR2 =  iTab;

    for( int i=0; i<256; i++ )
    {
        long aR = i*plates + (255-zR[i])*offset;
        long aR2 = i*plates + (255-zR2[i])*offset;
        long aG = i*plates + (255-zG[i])*offset;
        long aB = i*plates + (255-zB[i])*offset;
        long aC = i*plates + (255-zC[i])*offset;
        long aM = i*plates + (255-zM[i])*offset;
        long aY = i*plates + (255-zY[i])*offset;
        long aW = i*plates + (255-zW[i])*offset;
        long aW2 = i*plates + (255-zW2[i])*offset;
        long aW3 = i*plates + (255-zW3[i])*offset;

        pTarget[0+aR] = 255;
        pTarget[0+aR2] = 255;
        pTarget[0+aY] = 255;
        pTarget[0+aM] = 255;
        pTarget[0+aW] = 255;
        pTarget[0+aW2] = 255;
        pTarget[0+aW3] = 255;

        pTarget[1+aG] = 255;
        pTarget[1+aY] = 255;
```

```
            pTarget[1+aC] = 255;
            pTarget[1+aW] = 255;
            pTarget[1+aW2] = 255;
            pTarget[1+aW3] = 255;

            pTarget[2+aB] = 255;
            pTarget[2+aC] = 255;
            pTarget[2+aM] = 255;
            pTarget[2+aW] = 255;
            pTarget[2+aW2] = 255;
            pTarget[2+aW3] = 255;
        }
    }


    // get image statistics for adaptive processes
    void image::getStats()
    {
        // clear histogram
        for(int i=0; i<256; i++ )   hist[i]=0;

        long w0, w1, h0, h1;
        if      (autoExposure == 2 ) // 80% width/height, centered
        {
            w0 = width/10;
            w1 = width-w0;
            h0 = height/10;
            h1 = height-h0;
        }
        else if (autoExposure == 3 ) // 50% width/height, centered
        {
            w0 = width/4;
            w1 = width-w0;
            h0 = height/4;
            h1 = height-h0;
        }
        else
        {
            w0 = 0;
            w1 = width;
            h0 = 0;
            h1 = height;
        }

        // get histogram
        // unfortunately, an extra pass throught the image
        for( int lines=h0; lines<h1; lines++ )
        for( int pixels=w0; pixels<w1; pixels++ )
        {
            // get the current pixel values
            long addr = pixels*plates + lines*offset;
            long r = pSource[addr+0];
            long g = pSource[addr+1];
            long b = pSource[addr+2];

            long y = r<g ? r:g; y = b<y ? b:y;
            long x = r>g ? r:g; x = b>x ? b:x;

            // don't count clipped values
            if( x<255 ) hist[x]++;
            if( y>0   ) hist[y]++;
        }

        // calculate distribution function
```

```
    {
        long total=0;
        for( int i=0; i<256; i++ )
        {
            total += hist[i];
            dist[i] = total;
        }
        total /= 255;
        for( int i=0; i<256; i++ ) dist[i] /= total;
    }

    // calculate the inverse distribution function
    {
        for( int i=0, j=0;i<256; i++)
            while( (dist[i] >= j)) qist[j++] = i;
    }

    // use statistics to set the rerange table
    setRerangeTable();
}


void image::setRerangeTable()
{
    // set inflection points
    long inflect[] = {2,120,130,253};

    // Select inflection points
    long v0 = inflect[0];
    long v1 = inflect[1];
    long v2 = inflect[2];
    long v3 = inflect[3];

    // Use the inverse distribution function to estimate the slope of the dist function
    long q0 = qist[v0];
    long q1 = qist[v1];
    long q2 = qist[v2];
    long q3 = qist[v3];

    // estimate the black slope required to linearize the transfer function
    long mk = (255*(v1-v0))/(q1-q0);
    long mkMax = 3*255;
    long mkMin = 255/4;                    // too aggressive
         mkMin = (255*(q3-q0))/(v3-v0); // keep black gain
    mk = mk > mkMax ? mkMax:mk;            // limit the maximum slope
    mk = mk < mkMin ? mkMin:mk;            // limit the minimum slope

    // estimate the white slope required to linearize the transfer function
    long mw = (255*(v3-v2))/(q3-q2);
    long mwMax = 3*255;                    // too aggressive
         mwMax = (255*(v3-v0))/(q3-q0);
    long mwMin = 255/4;
    mw = mw > mwMax ? mwMax:mw;            // limit the maximum slope
    mw = mw < mwMin ? mwMin:mw;            // limit the minimum slope

    // convert the slopes to user value, pwl model
    long uk;
    if      ( mk>=255 )     uk = (mk-255)/2;
    else                    uk = (255*(mk-255))/(255-64);

    long uw;
    if      ( mw>=255 )     uw = -(mw-255)/2;
    else                    uw = -(255*(mw-255))/(255-64);
```

```
    // rerange
    long i0 = qist[2];
    long i1 = qist[253];

    for( int i=0;  i<i0;  i++ ) rrTable[i] = 0;
    for( int i=i0; i<i1;  i++ ) rrTable[i] = (255*(i-i0))/(i1-i0);
    for( int i=i1; i<256; i++ ) rrTable[i] = 255;

    // apply curves
    // this has a rerange error
    for( int i=0;  i<256;  i++ ) rrTable[i] = tt( uk, uw,  rrTable[i]);
}

void image::rerange(long *px, long *py, long *pz)
{
    // note that this is a repair process, not a controlled process!
    // the transforms do  not preserve color, but they undo prior exposure errors
    long x = *px;
    long y = *py;
    long z = *pz;
    long xx = rrTable[x];
    long yy = rrTable[y];
    long zz = rrTable[z]; //(yy*(x-y)  +  (xx-yy)*(z-y))/(x-y);
    *px = xx;
    *py = yy;
    *pz = zz;
}


// pixel transform, master algorithm, xyz is a permuted rgb space
// the following transforms should be in vx space !!
// these will not have full dynamic range, but the code is simpler and faster
inline void image::pixelTransform(long *xx, long *yy, long *zz, long *p, long x, long y, long
{
    long xxx = pa;
    // apply adaptive transforms if required
    if( autoExposure !=0)
    {
        rerange(&x,&y,&z);
        if( acuity !=0)     rerange(&xa,&ya,&za);
    }

    // convert vivid/gray and light/dark to bright/dkGray and deep/ltGray
    long umx = (ppf[midToneColor] + ppf[midToneLightness])/2;
    long umy = -(ppf[midToneColor] - ppf[midToneLightness])/2;

    // this non-linear selection function causes some gamut kinks!!
    long sx = (x*x)/255;
    long sy = ((255-y)*(255-y))/255;

    long ux =   ( -ppl[neutral][darkGrayNeutrals]        * (255-x)
                  +ppl[neutral][lightGrayNeutrals]       * (y)
                  +((ppf[deepColors]*(sy)  +  umx*(255-sy))    * (x-y))/255
                )/255;

    long uy =   ( -ppl[neutral][darkGrayNeutrals]        * (255-x)
                 +ppl[neutral][lightGrayNeutrals]        * (y)
                  +((-ppf[brightColors]*(sx)  +  umy*(255-sx))*  (x-y))/255
                )/255;

    *xx = tt(ux, ux, x);
    *yy = tt(uy, uy, y);
    *zz = zcalc(*xx, *yy, x, y, z, ppf[colorShift], &*p);
```

```
    if( acuity !=0)
    {
        long shsm = ppl[neutral][spatialAcuity];

        *xx = ((*xx)*255 + shsm*(*xx-xa))/255;
        *xx = *xx<0 ? 0:*xx;
        *xx = *xx>255 ? 255:*xx;

        *yy = ((*yy)*255 + shsm*(*yy-ya))/255;
        *yy = *yy<0 ? 0:*yy;
        *yy = *yy>255 ? 255:*yy;

        *zz = ((*zz)*255 + shsm*(*zz-za))/255;
        *zz = *zz<0 ? 0:*zz;
        *zz = *zz>255 ? 255:*zz;
    }
}

// permutation function, conveniently reorder rgb
inline void image::permute(long *p, long *x, long *y, long *z, long r, long g, long b)
{
    if(r>=g)    if(g>=b)                { *p = red;     *x = r;  *z = g;  *y = b; }
                else        if(b>=r)    { *p = blue;    *x = b;  *z = r;  *y = g; }
                            else        { *p = magenta; *x = r;  *z = b;  *y = g; }
    else        if(r>=b)                { *p = yellow;  *x = g;  *z = r;  *y = b; }
                else        if(g>=b)    { *p = green;   *x = g;  *z = b;  *y = r; }
                            else        { *p = cyan;    *x = b;  *z = g;  *y = r; }
    return;
}

// unPermute function, restore original order
inline void image::unPermute(long *r, long *g, long *b, long p, long x, long y, long z)
{
    switch(p)
    {
        case red:       *r=x;   *g=z;   *b=y;   break;
        case blue:      *r=z;   *g=y;   *b=x;   break;
        case magenta:   *r=x;   *g=y;   *b=z;   break;
        case yellow:    *r=z;   *g=x;   *b=y;   break;
        case green:     *r=y;   *g=x;   *b=z;   break;
        case cyan:      *r=y;   *g=z;   *b=x;   break;
    }
    return;
}

// calculate intermediate color value
inline long image::zcalc(long xx, long yy, long x, long y, long z, long U, long *p)
{
    if( y==x )  return xx;
    if(U==0)    return (yy*(x-y) + (xx-yy)*(z-y))/(x-y);
    else // shift the hue
    {
        U /=3;  // prevents hue kinks I think
        long u = (255*(z-y))/(x-y);
        if( sign[*p] >=0 )  u = u+U;
        else                u = u-U;
        if( u>255 )     { *p = next1[*p];   u = 510-u;  }// change d
        else if ( u<0 ) { *p = next0[*p];   u =  -u;    }// change d
        return (yy*255 + u*(xx-yy))/255;
    }
}


// calulate 5x5 neighborhood color values for sharpen/smooth
```

```
inline void  image::neighborhood(long *ra, long *ga, long *ba, long pixels, long lines)
{
#if 1 // normal version
    long r = 0, g = 0, b = 0;
    for( int j=-2; j<3; j++ )
    for( int i=-2; i<3; i++ )
    {
        long al = lines+j;  al = al<0 ? 0:al;   al = al>=height ? (height-1):al;
        long ap = pixels+i; ap = ap<0 ? 0:ap;   ap = ap>=width ? (width-1):ap;
        long aa = ap*plates + al*offset;
        r += pSource[aa+0]; g += pSource[aa+1]; b += pSource[aa+2];
    }
    *ra = r/25; *ga = g/25; *ba = b/25;
    return;
#endif

#if 0 // exclude clipped values version
    long r = 0, g = 0, b = 0, n=0;
    for( int j=-2; j<3; j++ )
    for( int i=-2; i<3; i++ )
    {
        long al = lines+j;  al = al<0 ? 0:al;   al = al>=height ? (height-1):al;
        long ap = pixels+i; ap = ap<0 ? 0:ap;   ap = ap>=width ? (width-1):ap;
        long aa = ap*plates + al*offset;

        long r0 = pSource[aa+0];
        long g0 = pSource[aa+1];
        long b0 = pSource[aa+2];

        if( r0 !=255 || g0 !=255 || b0 !=255 )
        {
            r += r0;    g += g0;    b += b0;    n++;
        }
    }
    if( n !=0 )
    {
        *ra = r/n;  *ga = g/n;  *ba = b/n;
    }
    else
    {
        long aa = pixels*plates + lines*offset;
        *ra = pSource[aa+0];
        *ga = pSource[aa+1];
        *ba = pSource[aa+2];
    }
    return;
#endif
}



// separate source and target buffers required if sharpen/smooth is selected
void  image::transform()
{
    // this is for rgb only
    if( plates != 3) return;

    // aquire and validate transform parameters
    translate();

    // this is real ugly. white point should be done inline instead of here
    if( rwp!=0 || gwp!=0 || bwp!=0 ) whitePointShift();

    // gather statistics if required

}
```

```
    if( autoExposure != 0) getStats();

    // main image processing loop
    for( int lines=0; lines<height; lines++ )
    {
        for( int pixels=0; pixels<width; pixels++ )
        {
            // get the current pixel values
            long addr = pixels*plates + lines*offset;
            long r = pSource[addr+0];
            long g = pSource[addr+1];
            long b = pSource[addr+2];

            // permute the colors
            long p,x,y,z;
            permute(&p,&x,&y,&z,r,g,b);

            // convert user parameters to per pixel parameters
            setParameters(x, y, z, p);

            // get and permutate the neighborhood values if necessary
            long ra, ga, ba;
            long pa,xa,ya,za;
            if( acuity != 0 )
            {
                neighborhood(&ra,&ga,&ba,pixels,lines);
                permute(&pa,&xa,&ya,&za,ra,ga,ba);
            }

            // construct a foreground/background parameter, if necessary
            if( foreground != 0 ) setFore(pixels, lines);

            // the master pixel transform xyz -> xx,yy,zz
            long xx, yy, zz;
            pixelTransform(&xx, &yy, &zz, &p, x, y, z, pa, xa, ya, za);

            // undo the permutation to restore rgb order
            long R, G, B;
            unPermute(&R,&G,&B,p,xx,yy,zz);

            // output RGB
            pTarget[addr+0]   = (U8)R;
            pTarget[addr+1]   = (U8)G;
            pTarget[addr+2]   = (U8)B;
        }
    }
    // diagnostic graph
    if( diags !=0 ) showTable();

}

// interleaved version
int main()
{

    long wi = 512;
    long he = 512;
    long pl = 3;
    long os = pl*wi;
    long bSize = wi*he*pl;

    unsigned char * pS = (unsigned char *)malloc( bSize );
    unsigned char * pT = pS;
```

```
    if( pS==0 )
    {
        printf("Unable to allocated buffer.\n");
        goto PAUSE;
    }

    // Create the image and log it
     image i1 = image(pS, pT, wi, he, os, pl);

    printf("\n processing ....\n");

    i1.readProcess();

    // test here to see if their is something to do...
    //if( wi != i1.reSize[0] )  { printf("wrong width\n"); goto PAUSE; }
    //if( he != i1.reSize[1] )  { printf("wrong height\n"); goto PAUSE; }

    FILE * iFile = fopen ("inputImage", "rb");
    if(iFile==NULL )
    {
        printf("Can't open inputImage.\n");
        goto PAUSE;
    }
    fread( pS, sizeof(char), bSize, iFile );
    fclose ( iFile );

    i1.transform();

    FILE * xFile = fopen ("outputImage.raw", "wb");
    if(xFile==NULL )
    {
        printf("can't create outputImage\n");
        goto PAUSE;
    }
    fwrite( pS, sizeof(char), bSize, xFile );
    fclose ( xFile );

PAUSE:
    printf("\n      ... processing complete. Enter a number to exit.\n");
    int temp;
    scanf("%d", &temp);
    printf("\n      ... exit\n");

}


void  image::readFogParamArray()
{
    FILE * pFile = fopen (PARAM, "rb");
    if( pFile==NULL ) goto ERROR;
    fread( &pp, sizeof(pp), 1, pFile);
    fclose( pFile );
    if (pFog.version != versionCode ) goto ERROR;
    return;

ERROR:
    printf( "...valid parameter file not found\n" );
    exit( 0 );
}

void image::clip( filterDEF *pf)
{
    // merge values
    for(int i=0; i<(ncolors-1); i++ )
```

```
    {
        // neutral axis (NP) color correction
            pf-> whiteNP            [i] += pf-> whiteNP         [ncolors-1];
            pf-> lightGrayNP        [i] += pf-> lightGrayNP     [ncolors-1];
            pf-> darkGrayNP         [i] += pf-> darkGrayNP      [ncolors-1];
            pf-> blackNP            [i] += pf-> blackNP         [ncolors-1];
        // saturated color correction
            pf-> colorShift         [i] += pf-> colorShift      [ncolors-1];
        // tone correction
            pf-> brightColors       [i] += pf-> brightColors    [ncolors-1];
            pf-> deepColors         [i] += pf-> deepColors      [ncolors-1];
            pf-> midToneColor       [i] += pf-> midToneColor    [ncolors-1];
            pf-> midToneLightness   [i] += pf-> midToneLightness[ncolors-1];
    }

    int count = sizeof(*pf)/sizeof(UVAL);    // size of the parameter array
    UVAL *aa = (UVAL *)pf;                   // directly address the parameter array

    // clip values to +-100 and rerange to +-255
    for(int i=0; i<count; i++ )
    {
        aa[i] = (255*aa[i])/100;
        if( aa[i]>255 ) aa[i]=255;
        if( aa[i]<-255 ) aa[i]=-255;
    }
}

// clip, merge and rerange parameters
void image::translate()
{
    clip( &pp.filterA );
    clip( &pp.filterB );
}

// calculate per pixel parameters from user parameters
// spatial variations need to be composed here
// interpolate color and composition
// pf is the primary color and sf is the secondary color
inline void image::setParameters( long x, long y, long z, long p )
{
    if ( compose == 3 )
    {
        for(int j=0; j<6; j++)
        for(int i=0; i<7; i++)
        {
            long v = (plist[i][j]*(255+fore) + plist1[i][j]*(255-fore))/510;
            v = v<-255 ? -255:v;
            v = v>255 ? 255:v;
            ppl[i][j] = v;
        }

        // per pixel white point
        rwp = (rwpA*(255+fore) + rwpB*(255-fore))/510;
        gwp = (gwpA*(255+fore) + gwpB*(255-fore))/510;
        bwp = (bwpA*(255+fore) + bwpB*(255-fore))/510;

    }
    for(int j=0; j<6; j++)
    {
        long vp = ppl[pf[p]][j];    // primary color contribution
        long vs = ppl[sf[p]][j];    // secondary color contribution
        if( x==y ) ppf[j] = (vp+vs)/2;
        else       ppf[j] = (vp*(x-z) + vs*(z-y))/(x-y);
    }
```

```
}
// End of parameter translation
// copyright 2003, 4G Color, All rights reserved
```